AD-A206 028

(2)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Harris Corporation, Harris Ada Compiler, Version 4.0, Harris H1200 (Host) and (Target), 880603W1.09058	5. TYPE OF REPORT & PERIOD COVERED 06 June 1988 to 06 June 1989
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA	15. SECURITY CLASS (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Harris Corporation, Harris Ada Compiler, Version 4.0, Wright-Patterson AFB, Harris H1200 under VOS, Version 7.1 (Host) and (Target), ACVC 1.9.

AVF Control Number: AVF-VSR-AVF-VSR-157.0788
88-03-02-HAR

Ada COMPILER VALIDATION SUMMARY REPORT:

Certificate Number: 880603W1.09058
Harris Corporation
Harris Ada Compiler, Version 4.0
Harris H1200

Completion of On-Site Testing:
06 June 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: Harris Ada Compiler, Version 4.0

Certificate Number: 880603W1.09058

Host:	Target:
Harris H1200 under VOS, Version 7.1	Harris H1200 under VOS, Version 7.1

Testing Completed 06 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

[illegible]

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Virginia L. Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

Ada Compiler Validation Summary Report:

Compiler Name: Harris Ada Compiler, Version 4.0

Certificate Number: 880603W1.09058

Host:	Target:
Harris H1200 under VOS, Version 7.1	Harris H1200 under VOS, Version 7.1

Testing Completed 06 June 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

[illegible]

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report ~~(VSR)~~ describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability ~~(ACVC)~~. An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 06 June 1988 at Ft. Lauderdale, FL.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures and Guidelines</u>.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect

because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Harris Ada Compiler, Version 4.0

ACVC Version: 1.9

Certificate Number: 880603W1.09058

Host Computer:

Machine:	Harris H1200
Operating System:	VOS, Version 7.1
Memory Size:	6 megabytes

Target Computer:

Machine:	Harris H1200
Operating System:	VOS, Version 7.1
Memory Size:	6 megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

. Capacities.

The compiler correctly processes tests containing loop statements nested to 65 levels, block statements nested to 65 levels, and recursive procedures separately compiled as subunits nested to 17 levels. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

. Universal integer calculations.

An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation processes 64-bit integer calculations. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

. Predefined types.

This implementation supports no additional predefined types in the package STANDARD. (See tests B86001C and B86001D.)

. Based literals.

An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation raises NUMERIC_ERROR during execution. (See test E24101A.)

. Expression evaluation.

Apparently no default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype. (See test C32117A.)

Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

This implementation uses no extra bits for extra precision and all extra bits for extra range. (See test C35903A.)

Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

Sometimes NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)}

. Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array type is declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array subtype is declared. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications during compilation. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Aggregates.

In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, all choices are evaluated before being checked for identical bounds. (See test E43212B.)

All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are not supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are not supported. (See tests A39005E and C87B62C.)

Length clauses with SIZE specifications for derived integer types are not supported. (See test C87B62A:)

- Pragmas.

The pragma INLINE is supported for procedures and functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

- Input/output.

The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D and CE2102E.)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, cannot be created in OUT_FILE mode, and cannot be created in IN_FILE mode. (See test EE3102C.)

Multiple internal files cannot be associated with the same external file. The proper exception is raised when multiple access is attempted. (See tests CE3111A..E (5 tests), CE3114B, CE3115A, CE2107A..I (9 tests), CE2110B, CE2111D, CE2110B, and CE2111H.)

An external file associated with more than one internal file cannot be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are deleted when they are closed. (See tests CE2108A and CE2108C.)

. Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 350 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 26 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	107	1046	1511	17	18	46	2745
Inapplicable	3	5	342	0	0	0	350
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	184	466	480	242	166	98	139	326	137	36	234	3	234	2745
Inapplicable	20	106	194	6	0	0	4	1	0	0	0	0	19	350
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 350 tests were inapplicable for the reasons indicated:

- C35702A uses SHORT_FLOAT which is not supported by this implementation.
- C35702B uses LONG_FLOAT which is not supported by this implementation.

- A39005B and C87B62A use length clauses with SIZE specifications for derived integer types or for enumeration types which are not supported by this compiler.

- A39005E and C87B62C use length clauses with SMALL specifications which are not supported by this implementation.

- A39005G uses a record representation clause which is not supported by this compiler.

- The following tests use SHORT_INTEGER, which is not supported by this compiler:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	

- The following tests use LONG_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

- C86001F redefines package SYSTEM, but TEXT_IO is made obsolete by this new definition in this implementation and the test cannot be executed since the package REPORT is dependent on the package TEXT_IO.

- C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

- CE2107A..I (9 tests), CE2110B, CE2111D, CE2111H, CE3111A..E (5 tests), CE3114B, and CE3115A are inapplicable because multiple internal files cannot be associated with the same external file. The proper exception is raised when multiple access is attempted.

- The following 285 tests require a floating-point accuracy that exceeds the maximum of 9 digits supported by this implementation:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 26 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B24009A	B24204A	B24204B	B24204C	B25002A
B2A003A	B2A003B	B2A003C	B33301A	B36002A
B37201A	B38003A	B38003B	B38009A	B38009B
B44001A	B64001A	B67001A	B67001B	B67001C
B67001D	B91003B	B95001A	B97102A	BC1303F
BC3005B				

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the Harris Ada Compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Harris Ada Compiler using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a Harris H1200 host and target operating under VOS, Version 7.1.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled on the Harris H1200, and all executable tests were linked and run on the Harris H1200. Results were printed from the host computer.

The compiler was tested using command scripts provided by Harris Corporation and reviewed by the validation team. The compiler was tested using the following options:

Option	Effect
-el	Produce error listing. Used on all tests.
-w	Suppress warning messages. Used on Class A, C, D, L and support tests.

Tests were compiled, linked, and executed (as appropriate) using a single host computer and a single target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Ft. Lauderdale, FL and was completed on 06 June 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

Harris Corporation has submitted the following Declaration of Conformance concerning the Harris Ada Compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: Harris Corporation
Ada Validation Facility: Ada Validation Facility, ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name:	Harris Ada Compiler	Version:	Version 4.0
Host Architecture ISA:	Harris H1200	OS&VER #:	VOS, Version 7.1
Target Architecture ISA:	Harris H1200	OS&VER #:	VOS, Version 7.1

Implementor's Declaration

I, the undersigned, representing Harris Corporation, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Harris Corporation is the ownr of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

[illegible] Date:

Harris Corporation
Wendell E. Norton, Director of Contracts

Owner's Declaration

I, the undersigned, representing Harris Corporation, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

[illegible] Date:

Harris Corporation
Wendell E. Norton, Director of Contracts

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Harris Ada Compiler, Version 4.0, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

     ...

     type INTEGER is range -8_388_608 .. 8_388_607;

     type FLOAT is digits 9 range
       -2#1000_0000000000_0000000000.0#E127 ..
        2#.1111111111_1111111111_1111111111_11111111#E127

     type DURATION is delta 2#1.0#E-14 range
       -2#1_0000000000_0000000000_000.0# ..
        2#1111111111_1111111111_111.11111111111111#;

     ...

end STANDARD;
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

(Appendix F of the Ada Reference Manual)

5.1 PROGRAM STRUCTURE AND COMPILATION

A "main" program must be a non-generic subprogram that is either a procedure or a function returning an Ada STANDARD.INTEGER (the predefined type). A "main" program cannot be a generic subprogram, or an instantiation of a generic subprogram.

5.2 PRAGMAS

5.2.1 Implementation-Dependent Pragmas

Pragma CONTROLLED is recognized by the implementation but has no effect in this release.

Pragma INLINE is implemented as described in Section 6.3.2 and Appendix B of the RM. This implementation expands recursive subprograms marked with the pragma up to a maximum nesting depth of 4. Warnings are produced for nesting depths greater than this or for bodies that are not available for inline expansion.

Pragma INTERFACE is recognized by the implementation and supports calls to C and FORTRAN language functions with an optional link name for the subprogram. The Ada specifications can be either functions or procedures. All parameters must have mode IN.

For C, the types of parameters and the result type for functions must be scalar, access, or the predefined type ADDRESS defined in the package SYSTEM. Record and array objects can be passed by reference using the ADDRESS attribute. The default link name is the symbolic representation of the simple name converted to lowercase. The link name of interface routines can be changed via the implementation-defined pragma **ext.rnal_name**.

For FORTRAN, all parameters are passed by reference; the parameter types must have the type ADDRESS defined in the package SYSTEM. The result type for a FORTRAN function must be a scalar type. Care should be taken when using tasking and FORTRAN functions. Since FORTRAN is not reentrant, it is recommended that an Ada controller task be used to access FORTRAN functions. The default link name is the symbolic representation of the simple name converted to uppercase. The link name of interface routines can be changed via the implementation-defined pragma **external_name**.

Pragma MEMORY_SIZE is recognized by the implementation but has no effect. The implementation does not allow the package SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling SYSTEM with altered values.

Pragma OPTIMIZE is recognized by the implementation but has no effect in this release.

Pragma PACK causes the compiler to choose a non-aligned representation for composite types. In the current release, it does not cause objects to be packed at the bit level.

Pragma STORAGE_UNIT is recognized by the implementation but has no effect. The implementation does not allow the package SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling SYSTEM with altered values.

Pragma SUPPRESS is recognized by the implementation and applies from the point of occurrence to the end of the innermost enclosing block. The double parameter form of the pragma, with a name of an object, type, or subtype is recognized, but has no effect.

Pragma SYSTEM_NAME is recognized by the implementation but has no effect. The implementation does not allow the package SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling SYSTEM with altered values.

5.2.2 Implementation-Defined Pragmas

Pragma EXTERNAL_NAME provides a method for specifying an alternative *link name* for variables, functions and procedures. The required parameters are the simple name of the object and a string constant representing the link name. Note that this pragma is useful for referencing functions and procedures that have had pragma interface applied to them, in such cases where the functions or procedures have link names that do not conform to Ada identifiers. The pragma must occur after any such applications of pragma interface and within the same declarative part or package specification that contains the object.

Pragma INTERFACE_OBJECT provides an interface to objects defined in foreign languages. This pragma has a required first parameter that is the simple name of an Ada variable to be associated with the foreign object. The optional second parameter is a string constant that defines the link name of the object. By default, the link name of the object is the symbolic representation of the simple name converted to lowercase. The variable declaration must occur before the pragma and both must occur within the same declarative part or package specification.

Pragma INTERFACE_COMMON_OBJECT provides an interface to objects defined in foreign languages as common blocks. Its semantics and syntax are identical to those of pragma **interface_object** except that the second parameter is required. The second parameter must be a string constant representing the link name of the common block as defined by the foreign language.

Pragma INTERFACE_MCOM_OBJECT provides an interface to monitor common objects as defined externally--through a foreign language or some other means. Its semantics and syntax are identical to those of pragma **interface_common_object** except for an additional third parameter. The required third parameter must be a string constant representing the name of the Monitor Common disc area. This can be a fully rooted or relative pathname or a VOS areaname. (Note: The VOS convention of automatically inserting the characters "M" before the final name is retained, therefore, the user should not specify these characters.) Specification of this pragma does not cause the Monitor Common disc area to be created.

Pragma SHARED_PACKAGE provides for the sharing and communication of library level packages. All variables declared in a package marked pragma **shared_package** (henceforth referred to as a shared package) are allocated in a VOS Monitor Common area that is created and maintained by the implementation. The pragma can only be applied to library level package specifications. Each package specification nested in a shared package will also be shared and all objects declared in the nested packages will reside in the same VOS Monitor Common area as the outer package.

The implementation restricts the kinds of objects that can be declared in a shared package. No unconstrained or dynamically sized objects can be declared in a shared package. No access type objects can be declared in a shared package. No explicit initialization of objects can occur in a shared package. If any of these restrictions are violated, a warning message is issued and the package is not shared. These restrictions apply to nested packages as well. Note that if a nested package violates one of the above restrictions, this prevents the sharing of all enclosing packages as well.

Task objects are allowed within shared packages, however, data defined within those tasks are not shared.

Pragma **shared_package** accepts an optional argument that, if specified, must be a string constant containing a blank separated list of VOS disc area control options as defined by the following:

- N=name, which specifies the name of the Monitor Common disc area to be created. If this parameter is not specified, the implementation will choose a VUE pathname and create the file under the .mcom HAPSE subdirectory on the host system.

- P=n, where n is the pack number used when the monitor common disc area is created.

- G=n, where n is the granule size in sectors used when the Monitor Common disc area is created.

- NR, which specifies that the monitor common area is to be Non-Resident, which is the default.

- RS, which specifies that the monitor common area is to be Resident.

- L=address, which specifies an address where the monitor common disc area is to be bound into physical memory. This is useful for sharing packages across systems configured with shared memory. Note that the RS control option must be specified if L=address is used.

With the valid application of pragma shared_package to a library level package, the following assumptions can be made about the objects declared in the package:

- The lifetime of such objects is greater than the lifetime defined by the complete execution of a single program.

- The state of such objects is not changed between invocations of programs that reference objects, except as defined by the recreation of such programs.

- A program that causes the creation of such an object during the elaboration of a shared package retains the state of the object, if it previously existed, except as defined by the recreation of such a program.

Programs that attempt to reference the contents of objects declared in shared packages that have not been implicitly or explicitly initialized are technically erroneous as defined by the RM (3.2.1--18). This implementation, however, does not prevent such references and, in fact, expects them.

Since packages that contain objects that are initialized are not candidates for pragma **shared_package**, the implementation suggests that programs be created for the sole purpose of initializing objects in the shared package.

Pragma SHARE_BODY is used to indicate a desire to share or not share an instantiation. The pragma can reference the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on/off for all instantiations of that generic, unless overridden by specific SHARE_BODY pragmas for individual instantiations. When it references an instantiated unit, sharing is on/off only for that unit. The default is to share all generics that can be shared, unless the unit uses pragma INLINE.

Pragma SHARE_BODY is only allowed in the following places: immediately within a declarative part, immediately within a package specification, or after a library unit in a compilation, but before any subsequent compilation unit. The form of this pragma is:

pragma SHARE_BODY (*generic_name*, *boolean_literal*)

Note that a parent instantiation is independent of any individual instantiation. Therefore, recompilation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.

Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

Pragma OPT_LEVEL controls the level of optimization performed by the compiler. This pragma takes one of the following as an argument: NONE, MINIMAL, GLOBAL, or MAXIMAL. The default is MINIMAL. NONE produces inefficient code but allows for faster compilation time. MINIMAL produces more efficient code with the compilation time slightly degraded. GLOBAL produces highly optimized code but the compilation time is significantly impacted. MAXIMAL is an extension of GLOBAL that can produce even better code but may change the meaning of the program. MAXIMAL attempts strength reduction optimizations that may raise OVERFLOW exceptions when dealing with values that approach the limits of the architecture of the machine.

In general, programs should be developed and debugged using OPT_LEVEL (MINIMAL), reserving GLOBAL and MAXIMAL for a thoroughly tested product.

The following optimizations are performed at the various levels.

OPT_LEVEL NONE:
- Short circuit boolean tests
- Use of machine idioms
- Literal pooling

OPT_LEVEL MINIMAL: (in addition to those done with NONE)
- Binding of intermediate results to registers
- Determination of optimal execution order
- Simplification of algebraic expressions
- Re-association of expressions to collect constants
- Detection of unreachable instructions
- Elimination of jumps to adjacent labels
- Elimination of jumps over jumps
- Replacement of a series of simple adjacent instructions by a single faster complex instruction
- Constant folding

OPT_LEVEL GLOBAL: (in addition to those done with MINIMAL)
- Elimination of unreachable code
- Insertion of zero trip tests
- Elimination of dead code
- Constant propagation
- Variable propagation
- Constraint propagation
- Folding of control flow constructs with constant tests

Elimination of local and global common sub-expressions
Move loop invariant code out of loops
Reordering of blocks to minimize branching
Binding variables to registers
Detection of uninitialized uses of variables

OPT_LEVEL MAXIMAL: (in addition to those done with GLOBAL)
Strength reduction
Test replacement
Induction variable elimination
Elimination of dead regions

5.3 IMPLEMENTATION-DEPENDENT ATTRIBUTES

HAPSE has defined the following two attributes for use in conjunction with the implementation-defined pragma shared_package.

P'LOCK for a prefix P that denotes a package
P'UNLOCK for a prefix P that denotes a package

These attributes are only applicable to packages that have had pragma shared package applied to them. The 'LOCK attribute defines a function that alters the "state" of the package to the LOCK state. The function has two optional parameters and returns a BOOLEAN result that has the value TRUE if a successful LOCK operation occurred or FALSE if the package was already LOCKed. The 'UNLOCK attribute defines a function that alters the "state" of the package to the UNLOCK state. It has no parameters and returns a BOOLEAN result that has the value TRUE if a successful UNLOCK operation occurred or FALSE if the package was already UNLOCKed.

The "state" of the package is only meaningful to the 'LOCK and 'UNLOCK attribute functions that set and query the state. A LOCK state *does not prevent concurrent access* to objects in the shared package. These attributes only provide indivisible operations for the set and test of implicit semaphores that could be used to control access.

The first parameter of the 'LOCK attribute function must be of the base type BOOLEAN and specifies whether to put the program into a sleep state until such time as the package becomes UNLOCKed, before executing the LOCK operation. This parameter is declared with a default value of FALSE, such that no sleep will occur unless explicitly specified by the user. The sleep state is induced through the VOS SWAIT service. Note that the sleep state will not be pre-empted by the implementations time-slice for tasks. Note that even if sleep is requested, this does not guarantee that the LOCK operation will be successful when it finally is attempted.

The second parameter must be of the base type INTEGER and represents the timeout period ir clock ticks, should the function be requested to sleep. This parameter defaults to zero, which represents no timeout.

o The pragma inline is implemented as described in Section 6.3.2 and Appendix B of the RM. This implementation expands recursive subprograms marked with the pragma up to a maximum nesting depth of 4. Warnings are produced for nesting depths greater than this or for bodies that are not available for inline expansion.

5.4 SPECIFICATION OF PACKAGE SYSTEM

```
package SYSTEM is
    type ADDRESS is private;
    type NAME is (narris_vue);

    SYSTEM_NAME  : constant NAME :=harris_vue;

    -- System-Dependent Constraints

    STORAGE_UNIT    : constant :=8;
    MEMORY_SIZE     : constant :=6_291_456;

    -- System-Dependent Named Numbers

    MIN_INT         : constant :=-8_388-608;
    MAX_INT         : constant :=8_388_607;
    MAX_DIGITS      : constant :=9;
    MAX_MANTISSA    : constant :=37;
    FINE_DELTA      : constant :=2.0**(-37);
    TICK            : constant :=0.01;

    --Other System-dependent Declarations

    subtype PRIORITY is INTEGER range 0 .. 23;

    MAX_REC_SIZE    : INTEGER :=32_767 *3;

private

    type ADDRESS is new INTEGER;

end SYSTEM;
```

5.5 RESTRICTIONS ON REPRESENTATION CLAUSES

5.5.1 Pragma PACK

Bit packing is not supported. Certain objects and components can be packed to the nearest whole STORAGE_UNIT.

5.5.2 Length Clauses

The specifications T'SIZE and T'SMALL are not supported.

5.5.3 Record Representation Clauses

Component clauses must be aligned on multiples of three STORAGE_UNIT boundaries.

5.5.4 Address Clauses

Address clauses and interrupts are not supported.

5.6 OTHER REPRESENTATION IMPLEMENTATION-DEPENDENCIES

Change of representation is not supported for record types.

The ADDRESS attribute is not supported for the following entities: static constants, packages, tasks, labels, and entries.

Machine code insertions are not supported.

5.7 CONVENTIONS FOR IMPLEMENTATION-GENERATED NAMES

There are no implementation generated names.

5.8 INTERPRETATION OF EXPRESSIONS IN ADDRESS CLAUSES

Address clauses and interrupts are not supported.

5.9 RESTRICTIONS ON UNCHECKED CONVERSIONS

The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

5.10 IMPLEMENTATION CHARACTERISTICS OF I/O PACKAGES

5.10.1 Implementation of Strings as Applied to External Files

Strings that contain names of external files are interpreted in the following manner for each of the respective external file environments.

VUE external files: filenames can be composed of up to 512 characters of the ASCII character set except for "/", ascii.nul, and non-printable characters. Further, the first character of a file must be alphanumeric, "." or "_". If the "/" character is encountered in a string, it is interpreted as a separator between filenames that specify VUE directories.

VOS external files: filenames are composed of a one to eight character qualifier plus a one to eight character areaname. The first character of the areaname must be alphabetic. The remaining characters comprising the areaname may be drawn from the following set of characters: A-Z, 0-9, :, #, -, /, . and " ". The qualifier portion of a filename is optional. If specified, it must be comprised of an account portion, a name portion, and an asterisk. The account portion can be null, or one to four characters from the following set: 0-9. The name portion can be null, or one to four characters from the following set: A-Z, 0-9. The name portion cannot be null if the account portion is not null. If lowercase letters are encountered in the string they are converted to uppercase.

5.10.2 Interpretation of Strings as Applied to Form Parameters

The OPEN and CREATE I/O procedures accept FORM parameters, in order to specify implementation dependent attributes of files. The HAPSE implementation supports the attributes described below. These attributes may be specified in any order. Blanks may be inserted between attributes, however none are required. No attribute can be specified more than once. All attributes must be specified in uppercase. These attributes are only applicable to CREATE calls. A form string passed to OPEN is ignored.

File Type Attributes

BL	Blocked file
UB	Unblocked file
RA	Random file

These attributes specify the VOS file type of a file to be created. UB is the default for all files. In general, the defaults should not be overridden for direct and sequential I/O.

Double Buffered Blocking

DB	Defines a BL type file as permanently double buffered

This attribute can only be specified if the file type is BL.

Directory Type

CD	The VOS directory entry for this file is to be kept resident
DD	The VOS directory entry is kept on disc

Access Parameters

PR	PUBLIC READ
PW	PUBLIC WRITE
PD	PUBLIC DELETE
AR	ACCOUNT READ
AW	ACCOUNT WRITE
AD	ACCOUNT DELETE
OW	OWNER WRITE
OD	OWNER DELETE

These attributes determine the access permissions associated with a file. The default access is OW OD. Note that if any access attributes are specified, then only the specified accesses will be granted (i.e., OW OD is not assumed).

File Definition Attributes

A=n	Access level, n = 0-15, VOS access required to access file
B=n	Blocking factor, where n is 1-7 sectors
C=n	Current size, where n is the number of sectors requested
E=n	Eliminate date, where n is the number of days before purging
G=n	Granule size, where n is the number of sectors per granule
M=n	Maximum size, n = number of sectors to which file may expand
P=n	Pack number, n = pack number of pack on which to create file
T=n	Type number, n = 0-7, provided for user file classification

No spaces are allowed between the attribute letter, the equal sign, and the integer value.

5.10.3 Implementation-Dependent Characteristics of DIRECT I/O

Instantiations of DIRECT_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating DIRECT_IO to provide an upper limit on the record size. In any case, the maximum size supported is 32_768 * 3 * STORAGE_UNIT bits. DIRECT_IO raises USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

5.10.4 Implementation-Dependent Characteristics of SEQUENTIAL I/O

Instantiations of SEQUENTIAL_IO use the value MAX_REC_SIZE as the record size (expressed in STORAGE_UNITs) when the size of ELEMENT_TYPE exceeds that value. For example, for unconstrained arrays such as a string where ELEMENT_TYPE'SIZE is very large, MAX_REC_SIZE is used instead. MAX_RECORD_SIZE is defined in SYSTEM and can be changed by a program before instantiating SEQUENTIAL_IO to provide an upper limit on the record size. In any case, the maximum size supported is 32_768 * 3 * STORAGE_UNIT bits. SEQUENTIAL_IO raises USE_ERROR if MAX_REC_SIZE exceeds this absolute limit.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A few of the values being substituted are built using the following variables:

```
C50 = "That_is_longer_than_the_255_max_for_file_base_name"
C10 = "./File_name_"
C11 = "./File_name1"
C12 = "./File_name2"
```

A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..498 =>'A', 499 =>'1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..498 =>'A', 499 =>'2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..249 | 251..499 =>'A', 250 =>'3')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..249 | 251..499 =>'A', 250 =>'4')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..496 =>'0', 497..499 =>"298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..493 =>'0', 494..499 =>"69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..249 =>'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..249 =>'A', 250 =>'1')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..479 =>' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	8_388_607
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	8_388_607
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	"./^BAD-CHARACTER"

Name and Meaning	Value
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	{C10} & {C50} & {C50} & {C50} & {C50} & {C50}
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10_000_000_000.0
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	{C11} & {C50} & {C50} & {C50} & {C50} & {C50}
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	{C12} & {C50} & {C50} & {C50} & {C50} & {C50}
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-8_388_608
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	8_388_607
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	8_388_608
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10_000_000_000.0
$MAX_DIGITS Maximum digits supported for floating-point types.	9

TEST PARAMETERS

Name and Meaning	Value
$MAX_IN_LEN Maximum input line length permitted by the implementation.	499
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	8_388_607
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	8_388_608
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 =>"2:", 3..496 =>'0', 497..499=>"11:")
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 =>"16:", 4..495 =>'0', 496..499=>"F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 =>'"', 2..498 =>'A', 499 =>'"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-8_388_608
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	SHORT_SHORT_INTEGER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFD#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- B28003A: A basic declaration (line 36) incorrectly follows a later declaration.
- E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);"; The Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.
- C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.
- C35502P: The equality operators in lines 62 and 69 should be inequality operators.
- A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.
- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.
- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

- C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.

- A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.

- BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.

- AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.